HOOVER: Distributed, Flexible, and Scalable Streaming Graph Processing on OpenSHMEM

Max Grossman[1], Howard Pritchard[2], Tony Curtis[3], Vivek Sarkar[4]

[1]Rice University

[2]Los Alamos National Laboratory

[3]Stony Brook University

[4]Georgia Institute of Technology

LA-UR-18-27888

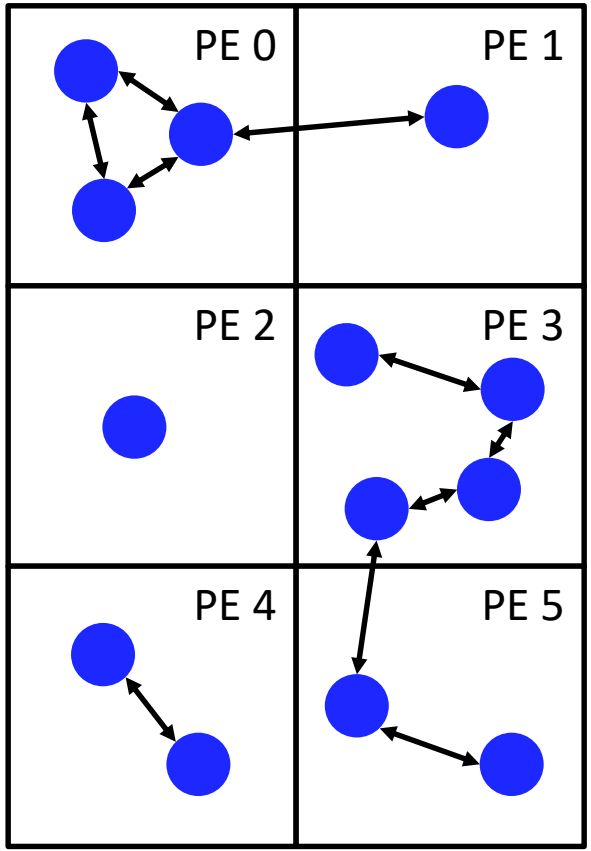
Target Class of Problems

Streaming, dynamic graphs with edge and vertex additions/removals.

Graph is naturally partitioned across PEs.

- Partitions of graph in different PEs have few cross-PE edges.
- PEs are naturally de-coupled by properties of the graph.

As graph evolves over time, connectivity may grow.



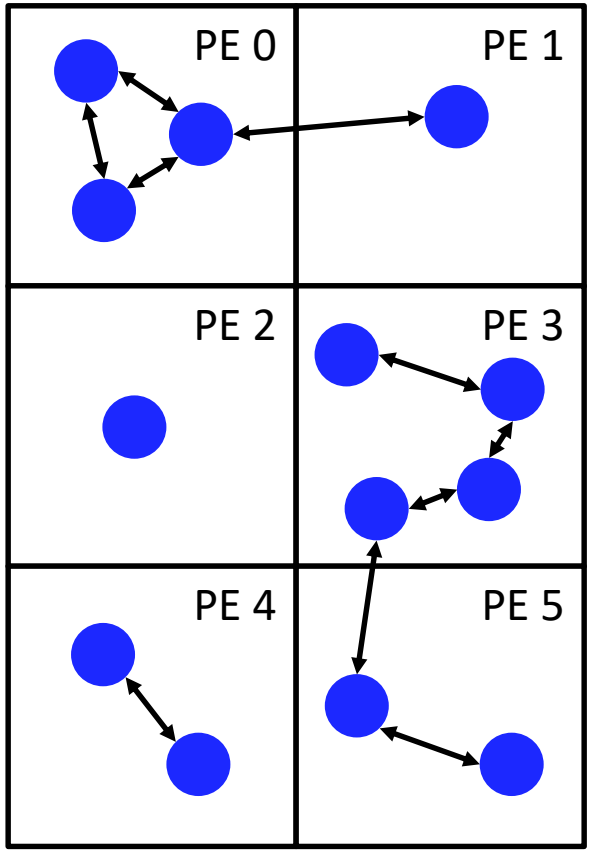
Illustrative Example

Financial fraud detection.

Vertices represent transactions.

Vertex attributes may be source acct, destination acct, amount, etc.

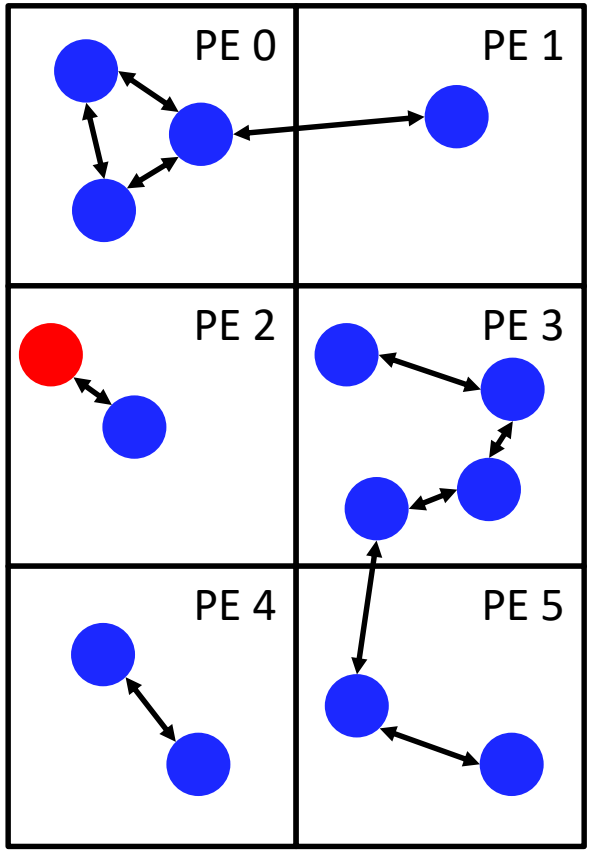
Streaming transactions for same account to same PE leads to natural partitioning – with a few super vertices (e.g. large vendors).



One Challenge With Distributed Graph Frameworks

New transaction arrives.

How would this be handled in existing graph frameworks?



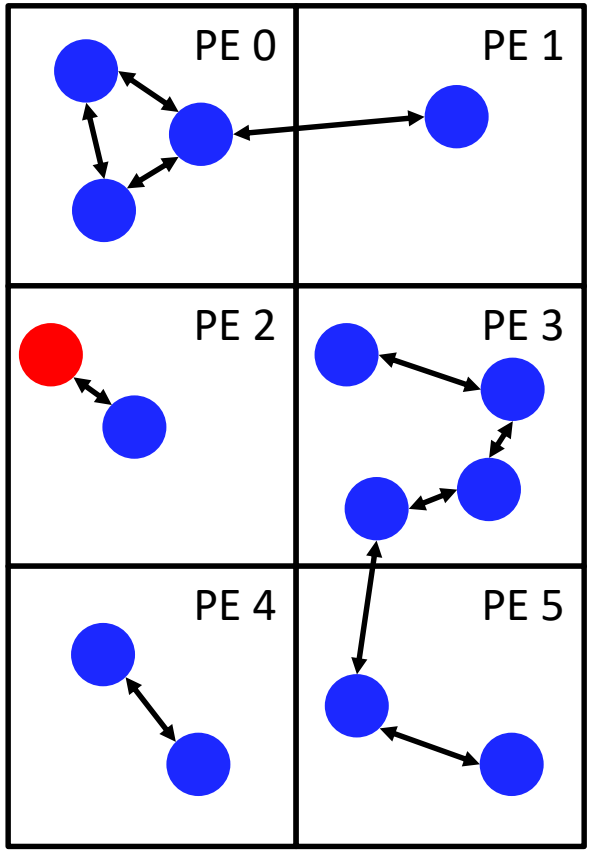
One Challenge With Distributed Graph Frameworks

New transaction arrives.

How would this be handled in existing graph frameworks?

GraphX (https://spark.apache.org/graphx/):

```
val transacts : RDD[(VertexID, ...)] = …
val new_transacts = transacts.flatMap(
  (vert) -> {
    if (insertNewVert) {
      return [vert, newVert()]
    } else {
      return [vert]
    }
  })
```



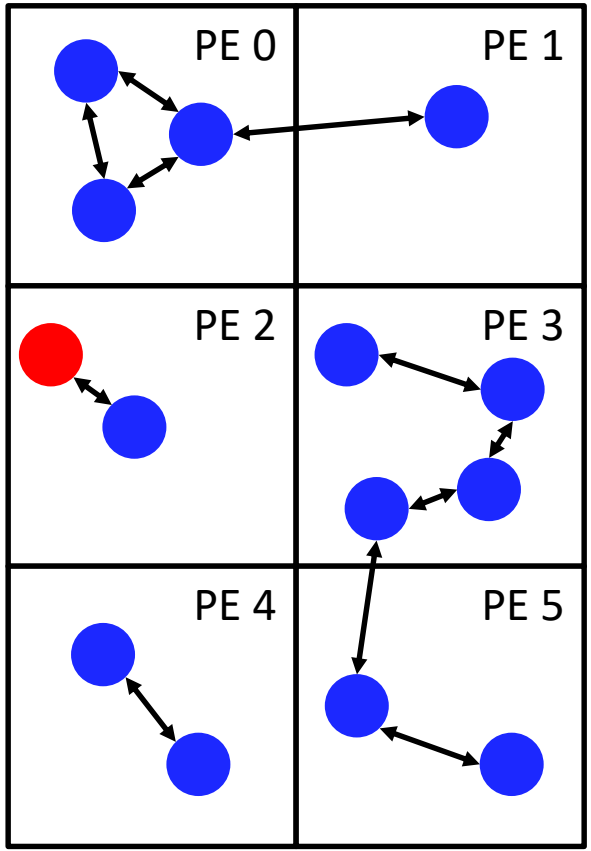
One Challenge With Distributed Graph Frameworks

New transaction arrives.

How would this be handled in existing graph frameworks?

GraphX (https://spark.apache.org/graphx/):

```
val transacts : RDD[(VertexID, ...)] = …
val new_transacts = sc.parallelize(local_new)
val next_transacts = transacts.join(new_transacts)
```



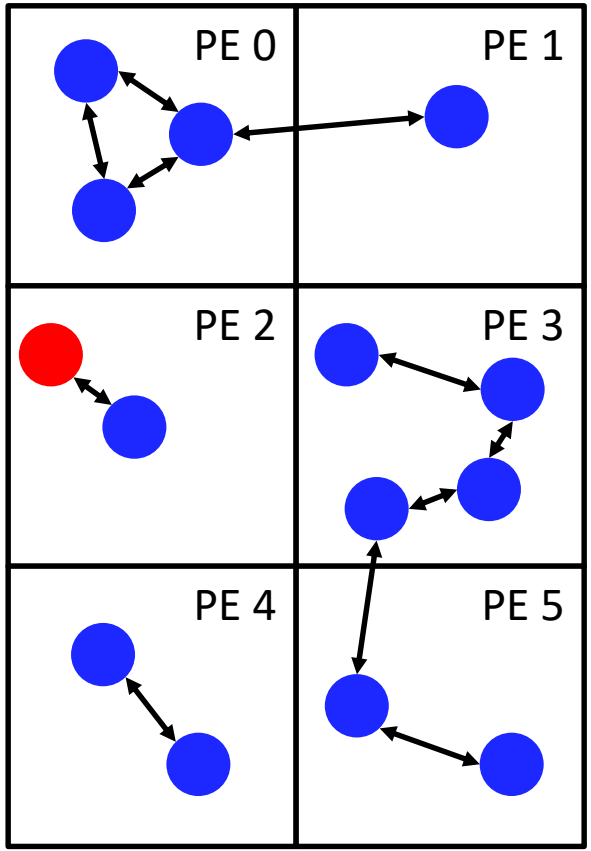
One Challenge With Distributed Graph Frameworks

GraphX (https://spark.apache.org/graphx/):

```
val transacts : RDD[(VertexID, ...)] = …
val new_transacts = sc.parallelize(local_new)
val next_transacts = transacts.join(new_transacts)
```

Problems:

- Both approaches are globally bulk synchronous, imply global barriers and possibly a shuffle.
- Only PE 2 really needs to be involved at this point – vertex insertion is entirely local.



Introducing HOOVER

Iterative dynamic graph modeling and analysis framework.

- Be able to update/mutate graphs
- Then analyze impact those updates have had on structure or other properties.

C/C++ library built on OpenSHMEM 1.4 – PGAS-by-design.

Emphasis on de-coupled execution – communication is one-sided and localized.

Runtime manages all computation and communication.

Users provide callbacks that implement application-specific functionality.

HOOVER sucking up your dynamic data…

HOOVER API

Vertices expose a sparse vector-like API.

APIs for creating, removing, updating vertices.

Edges are created/deleted implicitly based on vertex distance measures.

Vertices grouped into partitions (chunks of the problem space).

- Many more partitions than PEs

```
hvr_vertex_t *hvr_vertex_create_n(
    size_t nvecs, hvr_graph_id_t graph,
    hvr_ctx_t ctx);

void hvr_vertex_set(unsigned feature,
    double val, hvr_vertex_t *vec,
    hvr_ctx_t in_ctx);

double hvr_vertex_get(unsigned feature,
    hvr_vertex_t *vec, hvr_ctx_t in_ctx);

void hvr_vertex_delete_n(
    hvr_vertex_t *vecs, size_t nvecs,
    hvr_ctx_t ctx);
```

HOOVER API

Like other frameworks, callbacks are used to implement application-specific functionality.

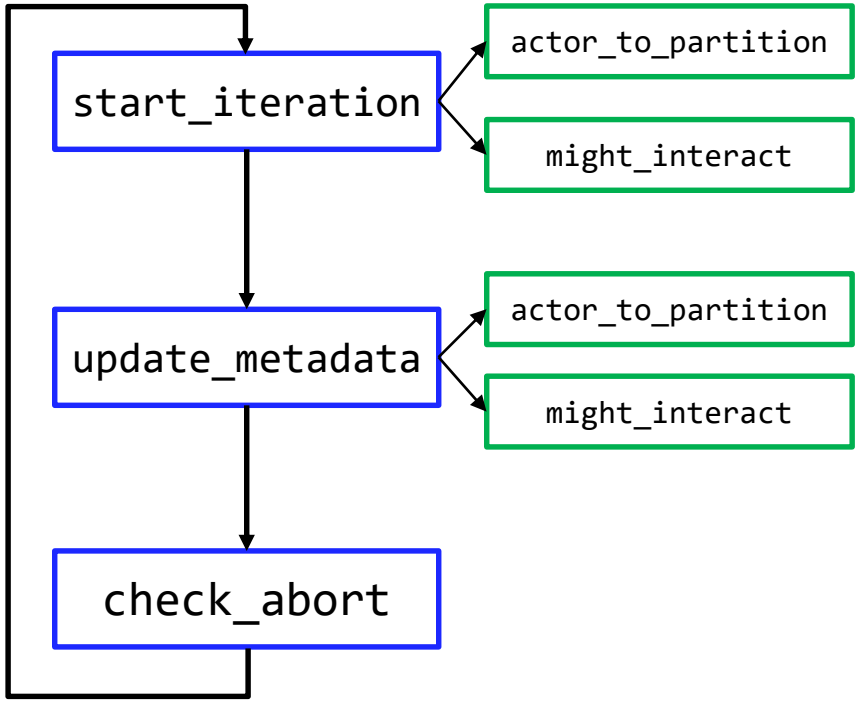
`start_iteration`: Hook for logic to be executed at the start of every iteration.

`update_metadata`: Given a vertex and its neighborhood, update its attributes.

`check_abort`: Called at end of iteration, check if this PE will exit.

`actor_to_partition`: Compute actor partition.

`might_interact`: Check if any actors in two given partitions might have an edge.

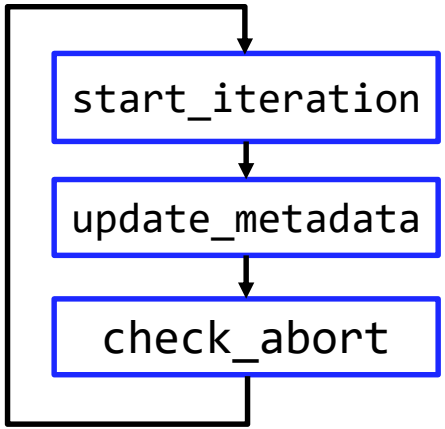


HOOVER API Example – Fraud Detection

Vertices represent transactions.

Vertex attributes may be source acct, destination acct, amount, etc.

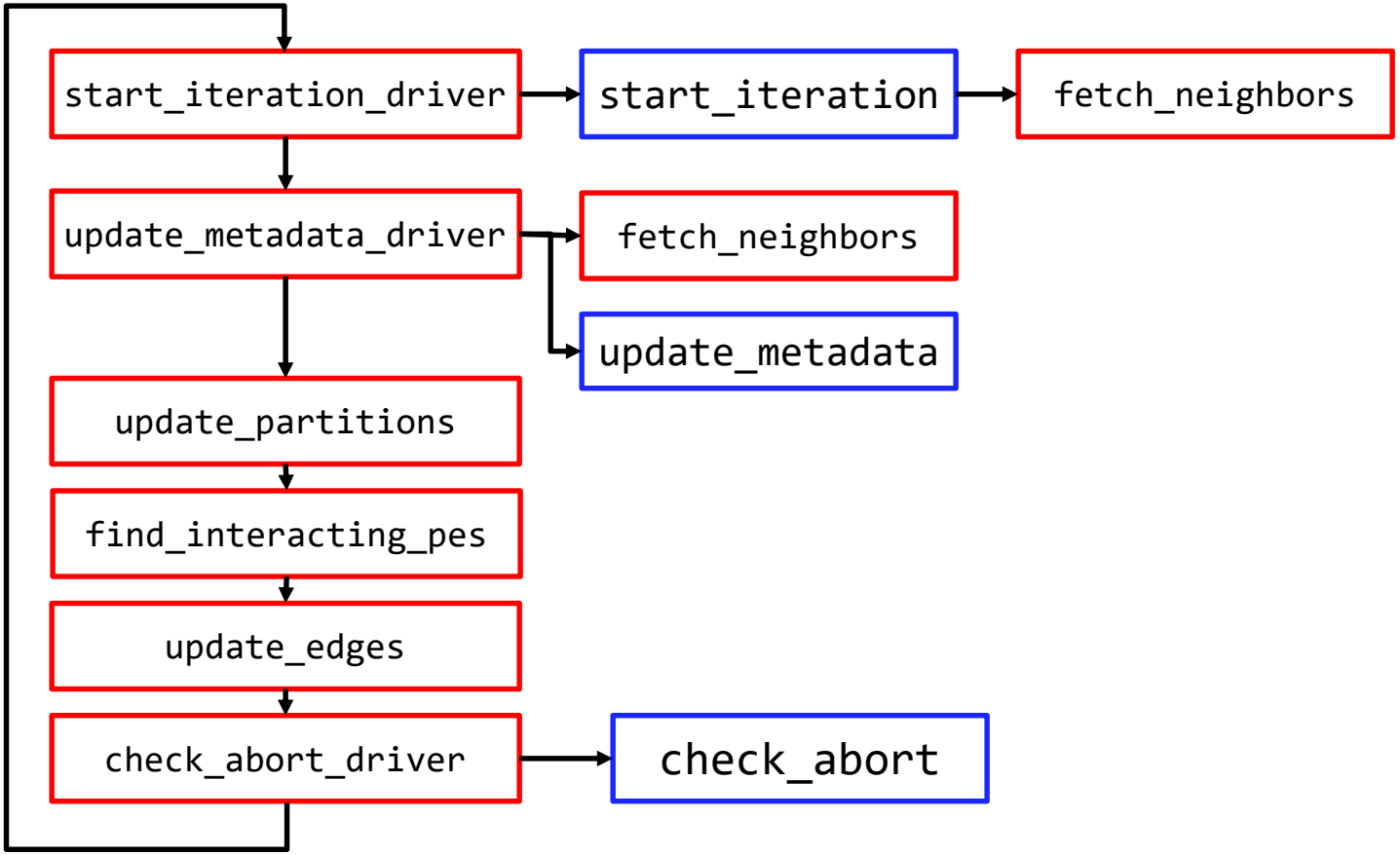
Edges represent similarities/relations between transactions.




```
start_iteration(vertex_iterator, ctx) {
  Injest data from external data streams;

  Convert into vertices in the graph;
}

update_metadata(vertex, neighbors) {
  Update per-vertex attributes for pattern matching;
}

check_abort(vertex_iterator, ctx) {
  Identify normative patterns;

  Identify anomalies based on global normative patterns;

  Print diagnostics, decide whether to exit;
}
```

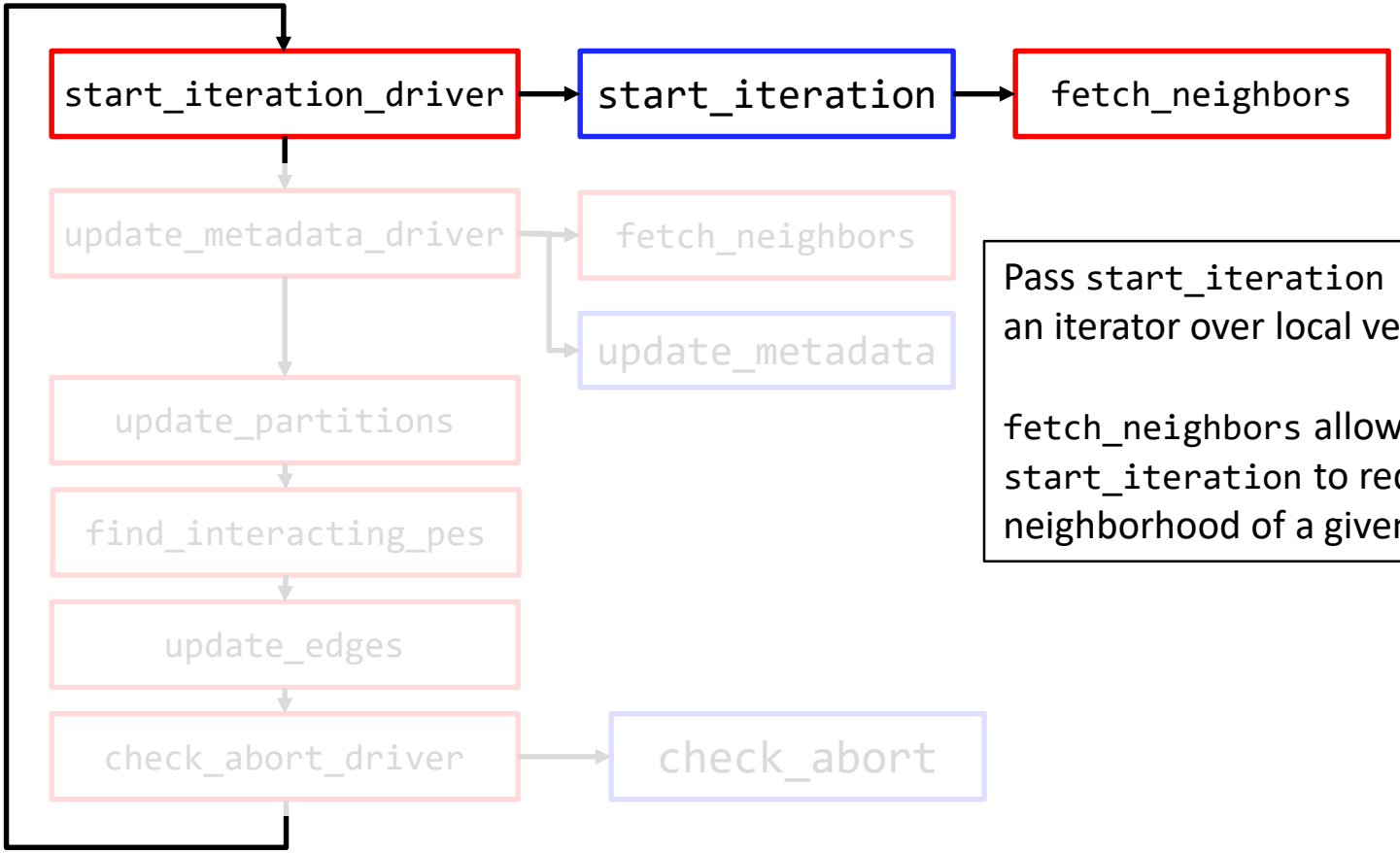
HOOVER Runtime

Responsible for coordination of all communication and computation.



HOOVER Runtime

Responsible for coordination of all communication and computation.

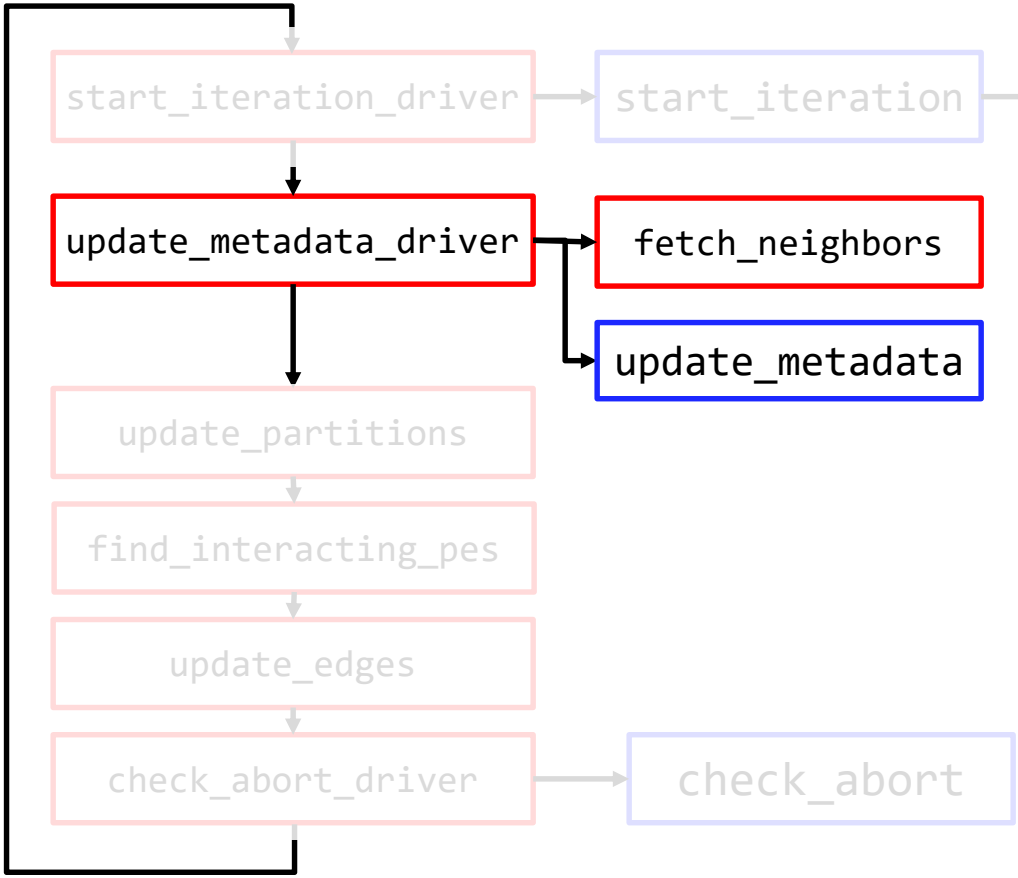


Pass `start_iteration` callback an iterator over local vertices.

`fetch_neighbors` allows `start_iteration` to request neighborhood of a given vertex.

HOOVER Runtime

Responsible for coordination of all communication and computation.



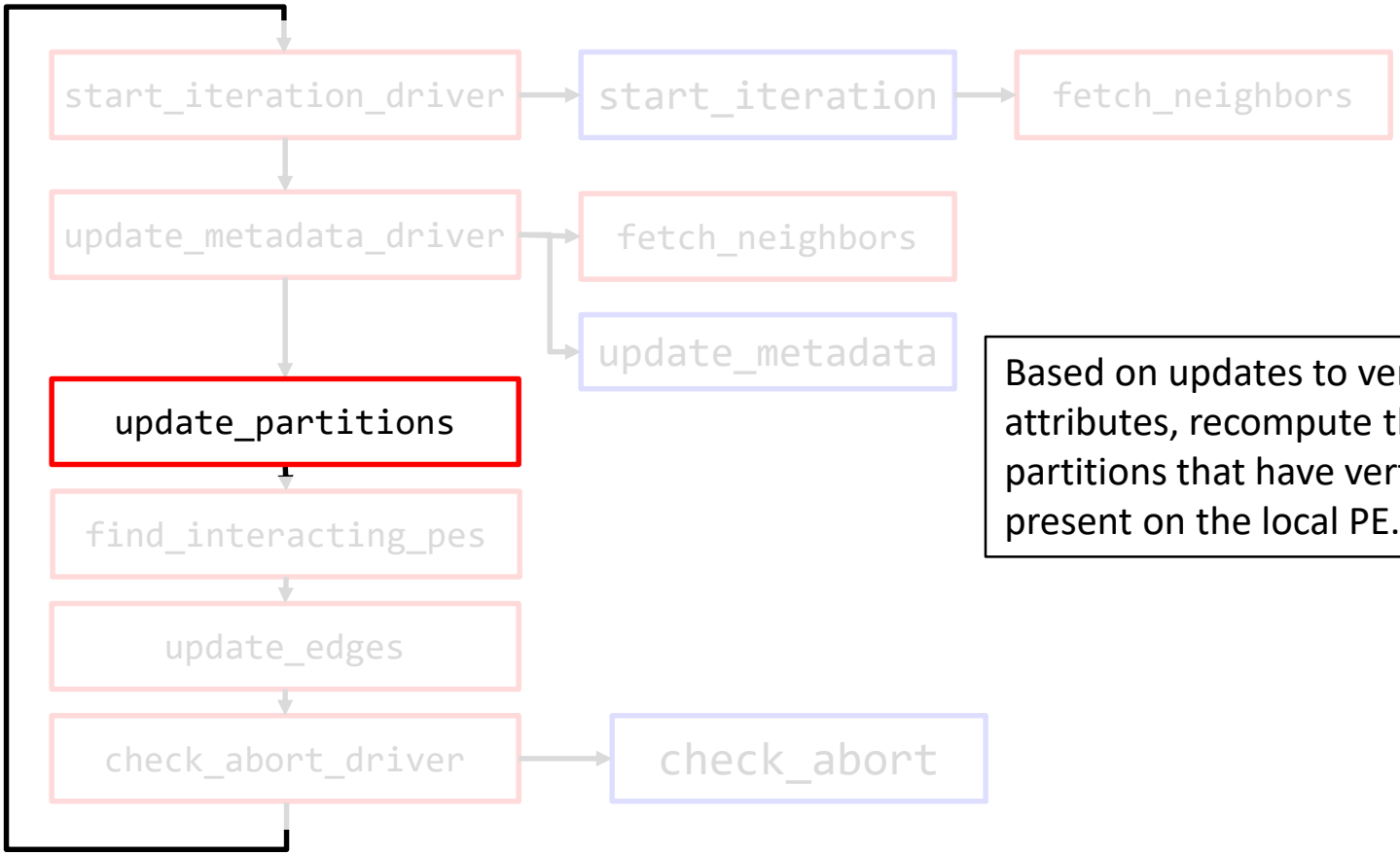
Iterate over each local vertex.

Use `fetch_neighbors` to collect its neighborhood.

Call `update_metadata` with the vertex and its neighborhood.

HOOVER Runtime

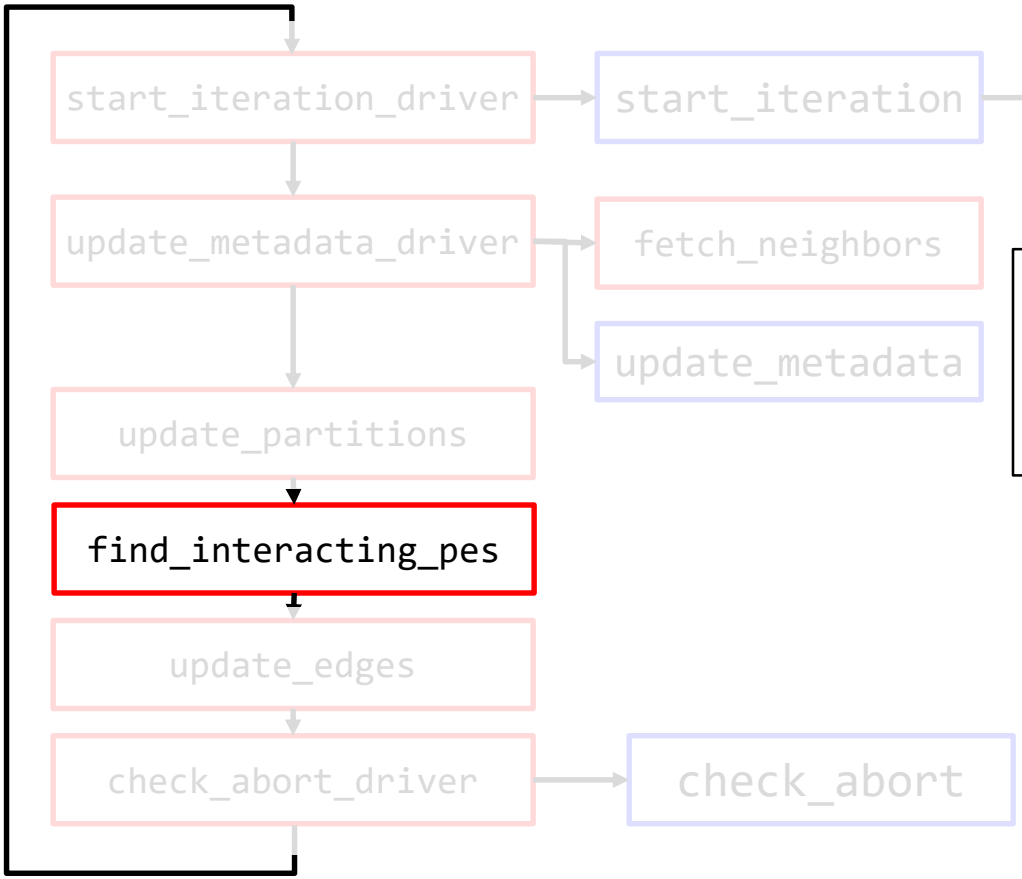
Responsible for coordination of all communication and computation.



Based on updates to vertex attributes, recompute the partitions that have vertices present on the local PE.

HOOVER Runtime

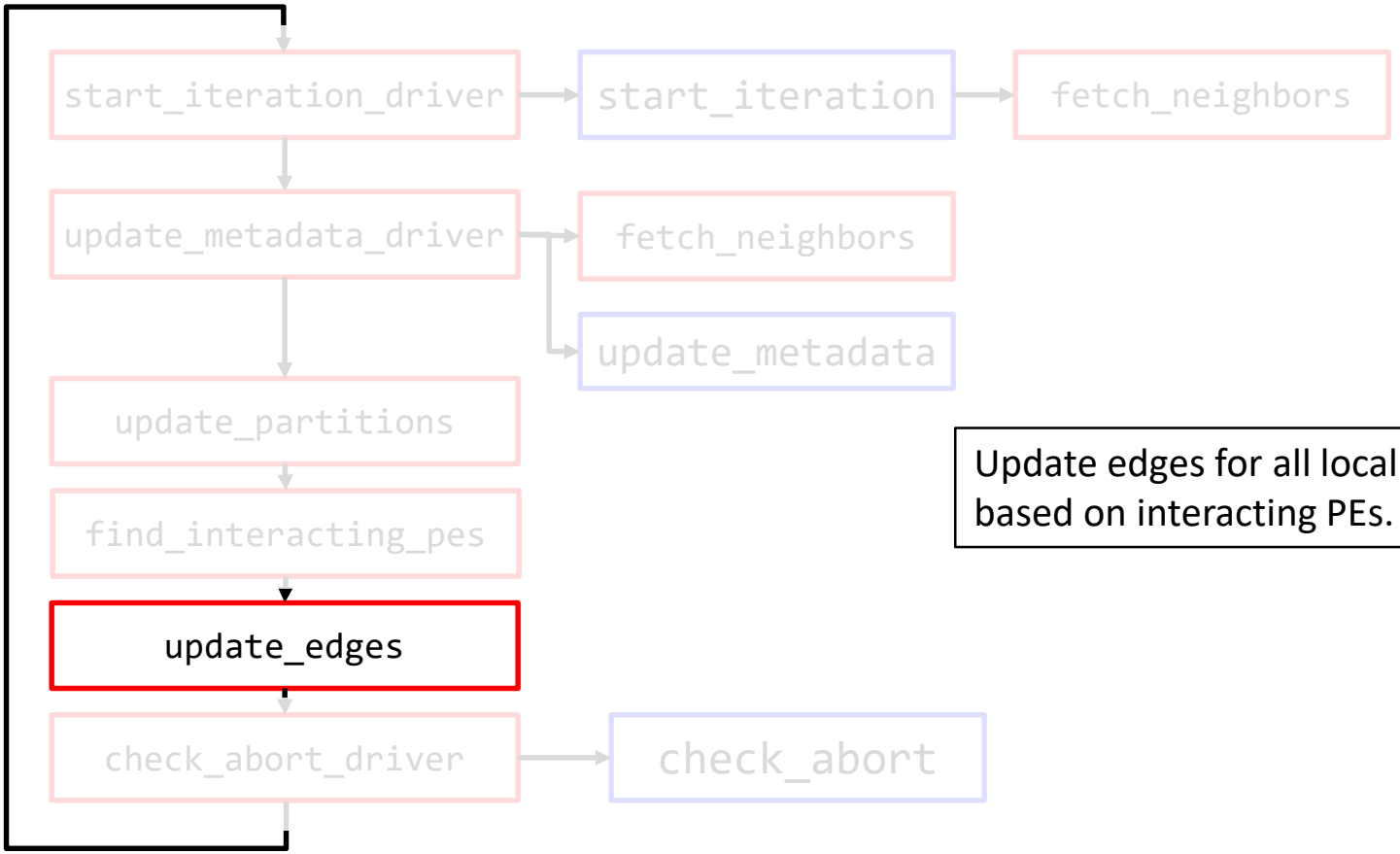
Responsible for coordination of all communication and computation.



Based on locally active partitions, find all other PEs containing vertices which my vertices may have edges with.

HOOVER Runtime

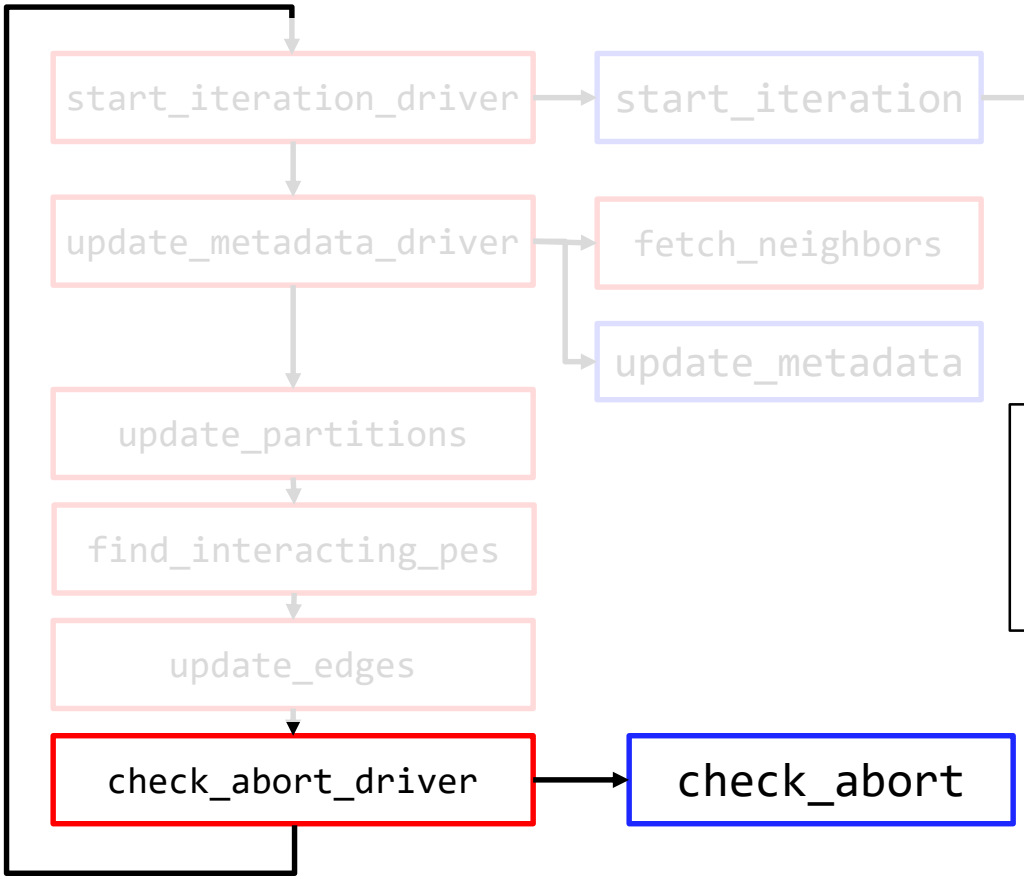
Responsible for coordination of all communication and computation.



Update edges for all local vertices based on interacting PEs.

HOOVER Runtime

Responsible for coordination of all communication and computation.



Based on all updates in current iteration, perform any cleanup and decide if we would like the current PE to exit the simulation.

Versioned Vertices

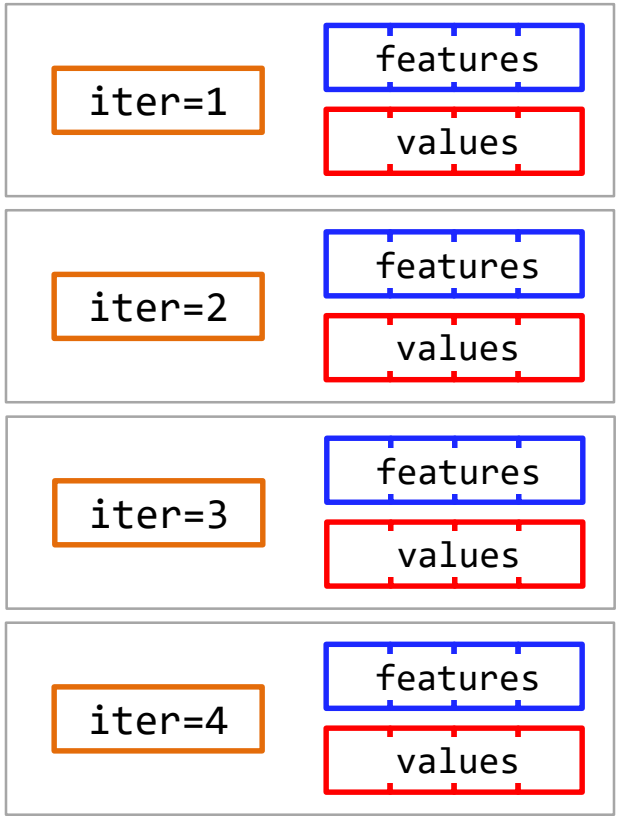
Decoupled nature of HOOVER requires remotely consistent vertices.

HOOVER is iterative -> sense of ordering between iterations.

PE A may access vertices on PE B when PE B is many iterations ahead of A in the simulation.

- Some applications may not care, and simply want the latest results.
- Others would like to prevent PE A from seeing into future iterations.

Every vertex maintains a history going back a fixed number of iterations.



Some HOOVER Optimizations

Remote Vertex Cache

Simple LRU fixed-size remote vertex cache.

Maximum age of members is tunable

- Clearing everything on every iteration ensures most up-to-date information, but increases communication.
- Not immediately evicting can lead to stale information.

Cache is bucketized by vertex ID to improve lookup speeds.

Read-Write Locks

Common communication pattern: atomically fetch large contiguous remote regions.

RW locks used to protect large memory regions, while enabling concurrent accesses.

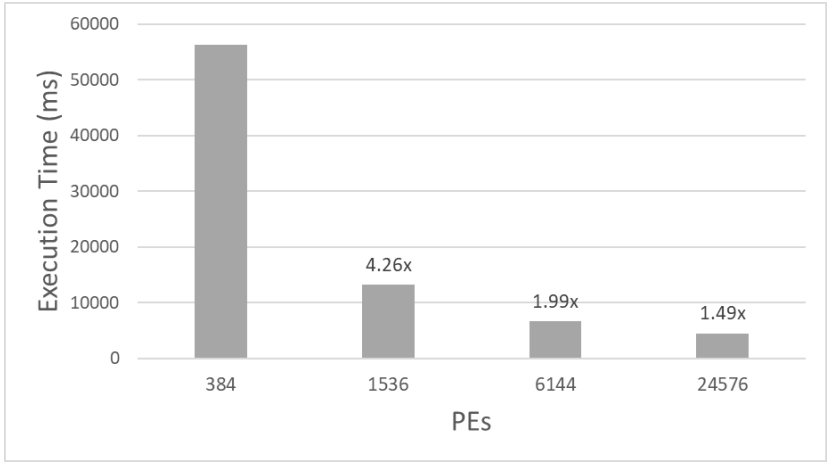
```
long *hvr_rwlock_create_n(const int n);

void hvr_rwlock_rlock(long *lock,
    int target_pe);
void hvr_rwlock_runlock(long *lock,
    int target_pe);
void hvr_rwlock_wlock(long *lock,
    int target_pe);
void hvr_rwlock_wunlock(long *lock,
    int target_pe);
```

Performance Evaluation

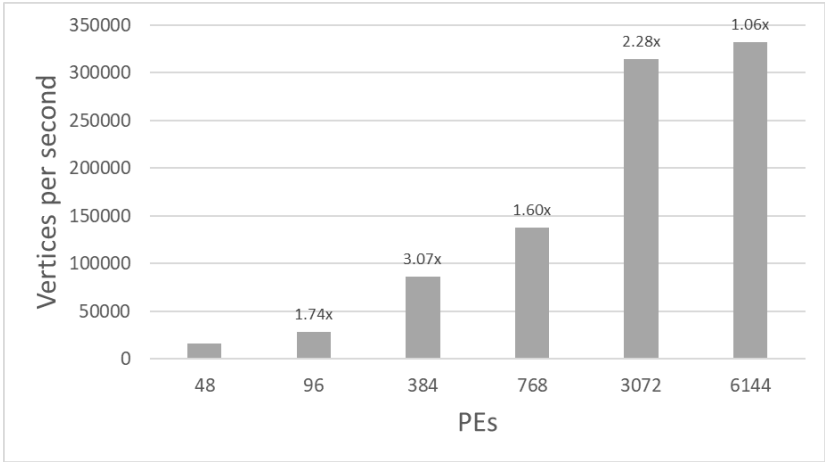
Experiments are run on Edison (CraySHMEM 7.7.0) – 1 PE per core (24 PEs per node).

Two applications developed from scratch:

- *Basic infectious disease model* – pool of infected/uninfected actors performing random walks.
- *Graph-based anomaly detection* – Stream random vertices into the graph, search for normative patterns, identify anomalies as patterns that are similar but not identical to normative.



Infectious disease model execution time scaling

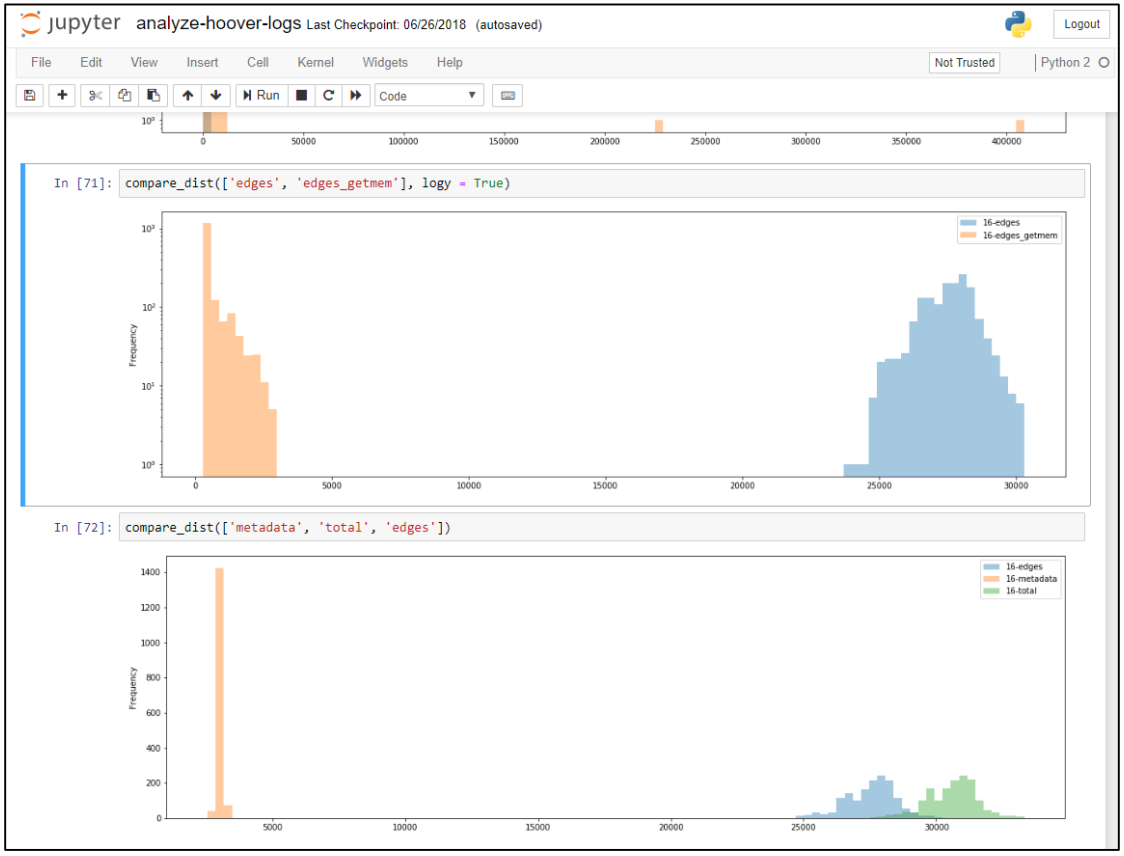


Graph-based anomaly detection processing rate scaling.

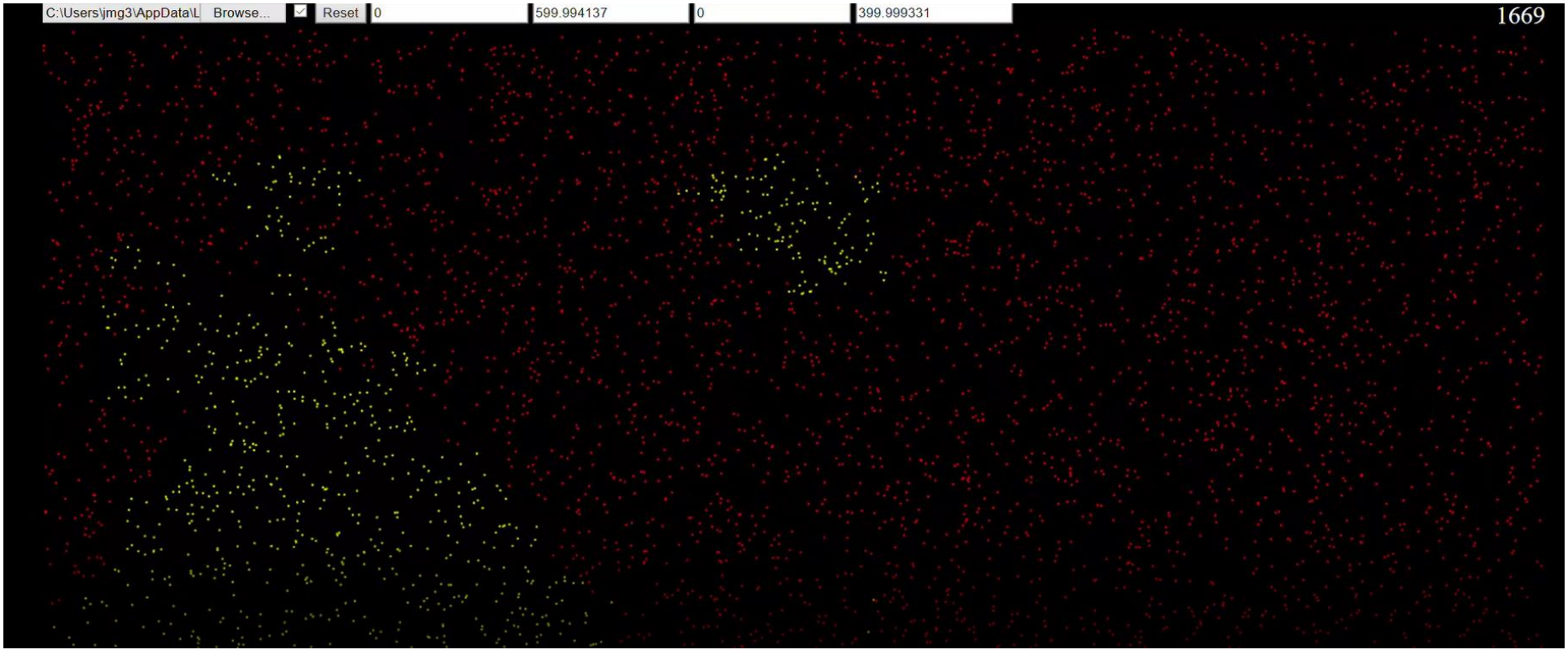
*Global barriers on each iteration reduce throughput to ~40% of HOOVER performance.

Performance Analysis w/ Jupyter

http://localhost:8888/notebooks/analyze-hoover-logs.ipynb



Simple Graph Viz



Ongoing Work

Wenbin Lu (SBU): Sophisticated infectious disease model for mosquito-borne illnesses (based on work by Manore et al).

Wes Suttle (SBU): Using HOOVER as a use case for exploring fault tolerance in OSSS OpenSHMEM.

Max Grossman (Rice):

- Continued API and performance improvements, primarily motivated by Wenbin's work on mosquito-borne illnesses and work on graph-based anomaly detection
- Cross-OpenSHMEM implementation performance comparisons
- Additional application development (Long Distance Leonard Jones)
- Multi-threading support
- Experiment with accelerators
- Experiment with active messages (supported by work by Jack Snyder, Duke University)

Conclusions

Introduced HOOVER: an iterative dynamic graph modeling and analysis framework.

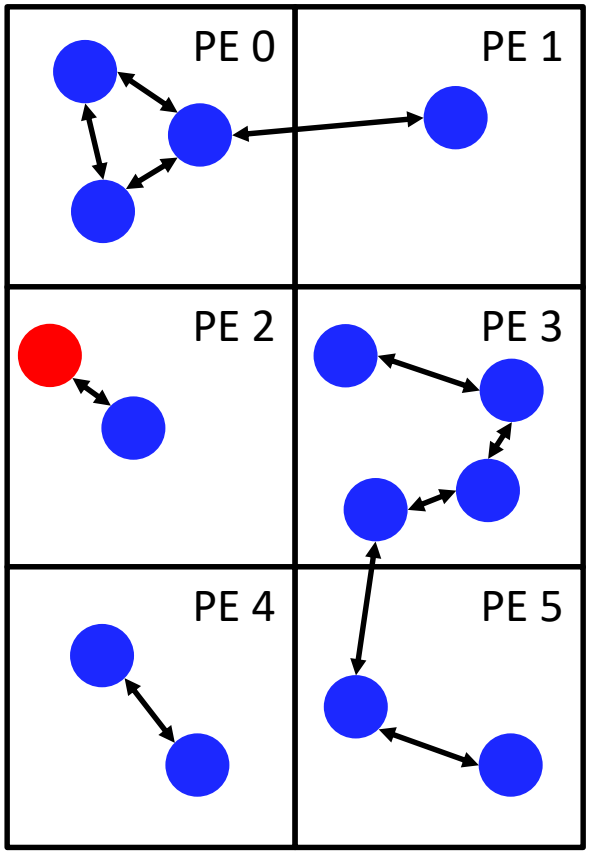
Emphasis on de-synchronized, de-coupled execution – one-sided and PGAS by default.

This adds complexity to the programming model and runtime.

- Remote consistency becomes a big and important challenge.

But enables scalability in a way that bulk synchronous models can't.

Github: https://github.com/agrippa/hoover
Contact: max.grossman@rice.edu



Acknowledgements

LA-UR-18-27888